INA: Assignment #2

Nejc Ločičnik, 63180183

University of Ljubljana — May 3, 2024

1 Where is SN100?

The dolphin network is a social network where we can look at links as social connections that exists between subjects/nodes. A subject/node only interacts with other subjects/nodes through available connections. Meaning we are looking for a node that acts as a bridge through which the subjects of the two communities interact with one another. Assuming that the interactions happen along the shortest path in the network, we are looking for a node that is part of that shortest path the most times. This is exactly what **betweenness centrality** does.

The implementation is very simple. Just use the networkx implementation of betweenness centrality and get the label of the node with the highest value.

```
G = u.read_pajek("dolphins.net")
u.info(G)
bet_cen = nx.betweenness_centrality(G)
G.nodes[max(bet_cen, key=bet_cen.get)]['label']
```

I additionally listed the other top 4 nodes with high betweenness centrality. It would be quite interesting to see if Beescratch also had enough influence to split the two communities in half, as it's betweenness centrality value is quite close to SN100. Maybe another dolphin could've been famous instead.

B. Centr:	Dolphin:
0.248	SN100
0.213	Beescratch
0.143	SN9
0.139	SN4
0.118	DN63

2 HIV and network sampling

To determine whether the original social graph and the sampled (15% of nodes sampled with a random walk) graph are small-world like and seemingly scale-free, we need to take a look at their statistics (table 1). A small-world graph requires a high clustering coefficient ($\langle C \rangle \gg 0$) and a short average shortest path length ($\langle d \rangle$), both of which are true for the original and sampled social graph. For the original social graph it could be argued that the clustering coefficient isn't that high (specially considering it's a social network, 0.268 might seem quite low), but I think if we take into consideration the graphs size relative to it's average degree, I find it reasonable that the clustering coefficient is on the lower end (plus small-world graphs aren't really strictly defined).

Graph	n	$\langle C \rangle$	$\gg \frac{\langle k \rangle}{n-1}$	$\langle d \rangle$	$\approx \frac{lnn}{ln\langle k \rangle}$
Social (full)	10680	0.268	0.0004	7.48	6.12
Social (sampled)	1602	0.445	0.0058	5.58	3.30

Table 1: Full and sampled social graph statistics.

To answer whether the original and sampled social graphs are seemingly scale-free, we need to look at their degree distributions shown in figure 1. As we can see both are quite straight (straight with a wide tail on the bottom right) and seem to follow the power-law distribution. The sampled graphs degree distributions might seem a bit iffy, but that is expected considering the smaller amount of sampled nodes. Both the original and sampled social graphs also contain hubs. The original graph has max degree of 205, while the sampled graph has a max degree of 170.

To summarize, both graphs are small-world like and seemingly scale-free.

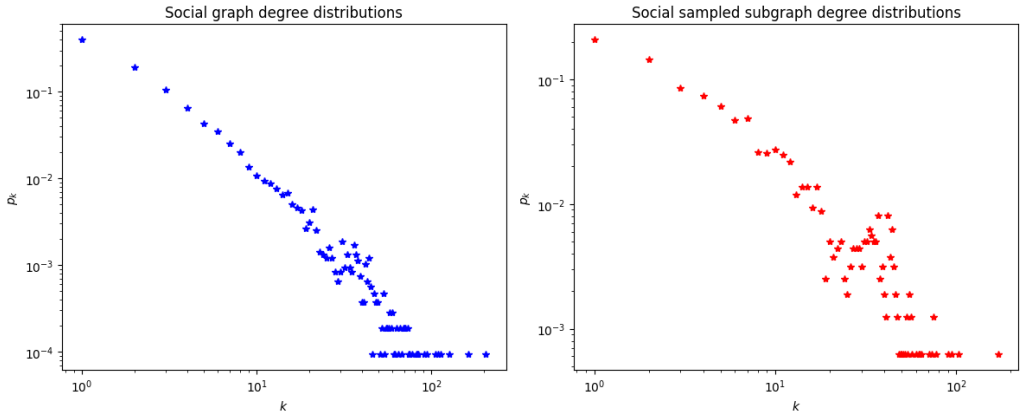

Figure 1: Degree distribution for full and sampled social graph.

Regarding why that is. First of all the original social graph is a real social network, meaning that the likelihood that it will exhibit properties of a small-world and a scale-free graph is high (just a general observation of real social networks).

Regarding why the sampled graph exhibits these properties as well, it's related in how we sampled it. Since we sampled using a random walk (and my implementation allows repeats, as in we can walk through a before visited node multiple times), which chooses the next step based on the available nodes (neighbours, meaning it depends on node degrees). Random walk will therefore naturally gravitate towards nodes with higher degrees (hubs, since they have a higher probability of being chosen), which is kind of similar to preferential models, which are based on a property called preferential attachment. These type of models are known to model a scale-free structure.

The random walk is also the reason for the small-world like structure. Since it gravitates towards hubs, my implementation allows walking through the same node multiple times and high degree nodes are generally connected to other high degree nodes, which means that we will mostly sample the hubs surroundings. The hubs surroundings should be well connected so there is a high chance that the sampled graph will retain short paths and a lot of triangle like structures, which is seen as the clustering coefficient is almost double the original graphs.

```
G = u.read_pajek("social.net")
u.info(G)
aspl = nx.average_shortest_path_length(G)
n = len(G)
k = sum(dict(G.degree).values())/n
print("Average shortest path | ", aspl)
print("approx | ", (np.log(n)/np.log(k)))
print(4.55/(10680-1))

current_node = random.choice(list(G.nodes))
visited_nodes = {current_node}

while True:
    if len(visited_nodes) == int(len(G.nodes)*0.15):
        break
    neighbours = list(G.neighbors(current_node))
    next_node = random.choice(neighbours)
    visited_nodes.add(next_node)
    current_node = next_node

sub_G = G.subgraph(visited_nodes)
u.info(sub_G)
aspl = nx.average_shortest_path_length(sub_G)
n = len(sub_G)
k = sum(dict(sub_G.degree).values())/n
print("Average shortest path | ", aspl)
print("approx | ", (np.log(n)/np.log(k)))
print(9.32/(1602-1))

def plot(G, ax, c, type=0):
    nk = {}

    for _, k in G.degree():
        if k not in nk:
            nk[k] = 0
        nk[k] += 1
    ks = sorted(nk.keys())

    ax.loglog(ks, [nk[k] / len(G) for k in ks], '*', color=c)
    ax.set_xlabel('$k$')

fig, (ax1, ax2) = plt.subplots(1, 2, tight_layout=True)
fig.set_figheight(5)
fig.set_figwidth(12)
plot(G, ax1, 'b', 0)
ax1.set_title("Social graph degree distributions")
ax1.set_ylabel('$p_k$')

plot(sub_G, ax2, 'r', 0)
ax2.set_title("Social sampled subgraph degree distributions")
ax2.set_ylabel('$p_k$')

plt.show()
```

3 Ring graph modularity

Some ring graph properties we can directly observe:

- The number of nodes is equal to the number of edges: $n = m$
- Each node has a degree of 2. Can also be calculated with n and m: $k_i = \langle k \rangle = \frac{2\cancel{m}}{\cancel{n}} = 2$
- Every cluster will be the same (amount of nodes, edges; it's structure).

The starting formula from the lecture slides. The second part is easier to work with.

$$Q = \frac{1}{2m} \sum_{ij} (A_{ij} - \frac{k_i k_j}{2m}) \delta c_i c_j = \sum_{C} \frac{m_c}{m} - (\frac{k_c}{2m})^2$$

- Since every cluster will be the same we can change the summation over clusters ($\sum_C$) into a simple multiplication ($C\cdot$)
- m_c represents the amount of edges inside the cluster, since a cluster alone represents a path graph, the amount of edges inside is: $m_c = n_c - 1$
- k_c represents the amount of degrees in the cluster (degrees representing edges connecting outside the cluster count and some edges will naturally be counted twice). Since every node has the same amount of degrees: $k_c = \langle k \rangle \cdot n_c = 2n_c$
- The amount of clusters C needs to be represented with n and n_c so: $C = \frac{n}{n_c}$

Including all the points above into the formula:

$$Q = \frac{n}{n_c} \cdot \left(\frac{(n_c - 1)}{n} - \left(\frac{\cancel{2}n_c}{\cancel{2}n} \right)^2 \right) = \frac{n}{n_c} \cdot \left(\frac{(n_c - 1)}{n} - \left(\frac{n_c}{n} \right)^2 \right)$$

The following is simply cleaning up the formula so that the next part of the exercise is simpler.

$$Q = \frac{\cancel{n} \cdot (n_c - 1)}{n_c \cdot \cancel{n}} - \frac{\cancel{m} \cdot \cancel{n_c} \cdot n_c}{\cancel{n_c} \cdot \cancel{m} \cdot m} = \frac{\cancel{n_c}}{\cancel{n_c}} - \frac{1}{n_c} - \frac{n_c}{n} = 1 - \frac{1}{n_c} - \frac{n_c}{n}$$

We now need to find the size of clusters n_c that maximizes modularity Q: $\frac{\partial Q}{\partial n_c} = 0$. Since we cleaned up the formula we can derive separately:

$$0 + \frac{1}{n_c^2} - \frac{1}{n} = 0 \rightarrow \frac{1}{n_c^2} = \frac{1}{n} \rightarrow n_c^2 = n \rightarrow n_c = \sqrt{n}$$

When n is a perfect square (square number) we have a single optimal solution ($n = 100, n_c^{opt} = 10$), seen in figure 2. In other cases (non-square numbers) we will have 2 optimal solutions for cluster size n_c which are the two closest factors of n. So we need to additionally round the $\sqrt{n}$ to the closest factor of n, seen in figure 3. This also holds when n is a prime number (only has 2 factors, 1 and itself), the modularity Q for $n_c = 1$ and $n_c = n$ will be the same, seen in figure 4.

The ring graph is clearly a toy example. The concept of "community" isn't as meaningful as in other types of graphs, since the graph is highly symmetric and each node has the same degree. Looking purely at the optimization of modularity Q, it's also not that meaningful considering we will have multiple solutions whenever n isn't a perfect square number. The resulting partition in those cases will not be unique.

Since modularity Q is used in community detection algorithms, we would like to have a single optimal solution, which would always produce a clear best and unique graph partition. This means that the detection algorithms would always produce the same partition (deterministic), unlike how they often produce different results.

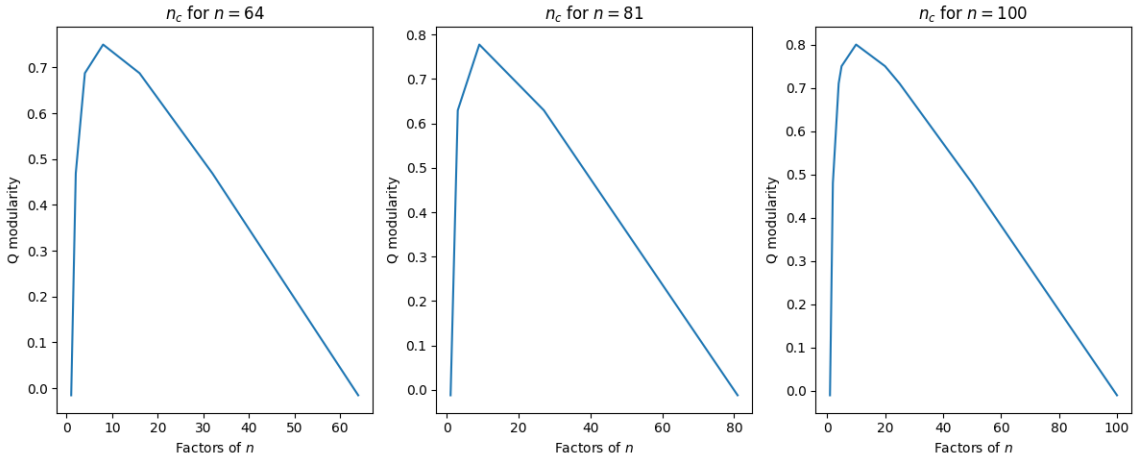

Figure 2: Optimal n_c when n is a perfect square number (single solution: $n_c = \sqrt{n}$).

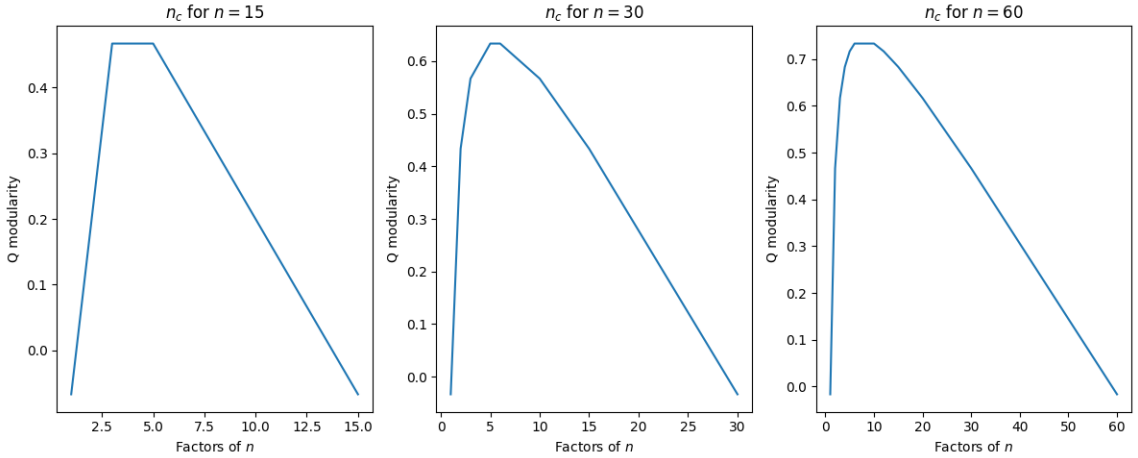

Figure 3: Optimal n_c when n is a non-square number (two solutions: 2 closest factors of n to $\sqrt{n}$).

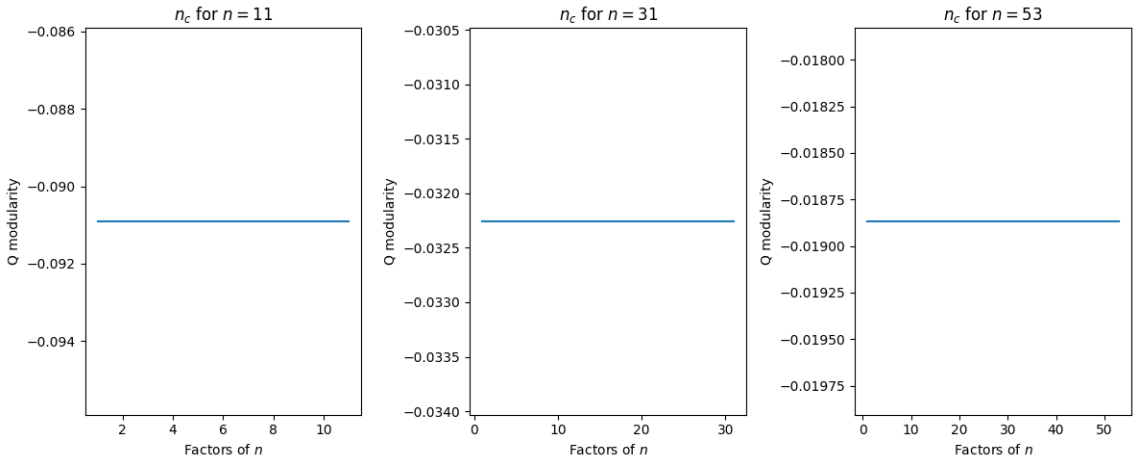

Figure 4: Optimal n_c when n is a prime number (two solutions: 1 and n).

4 Who's the winner?

4.1 Girvan-Newman benchmark graphs

The community detection algorithm that came out on top is Leiden, which only started failing when the mixing parameter μ was increased from 0.4. Infomap and fast label propagation performed almost the same, as their performance started dropping when the mixing parameter was increased from 0.2 onwards (figure 5).

The results seem as expected, Leiden is known for high accuracy, even in challenging scenarios. Infomap, which is based on minimizing the expected description length of a random walk and FLPA, a greedy algorithm that optimizes modularity are expected to have difficulty accurately detecting communities as they become more intermixed. The Girvan-Newman graphs at different μ breakpoints are visualized in figure 6.

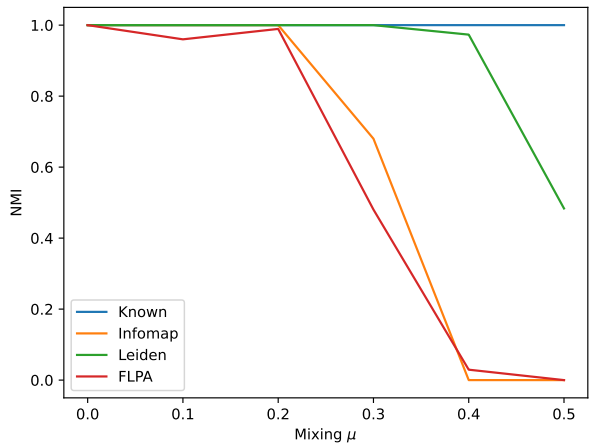

Figure 5: The accuracy of community detection algorithms (Infomap, Leiden, FLPA) on Girvan-Newman benchmark graphs.

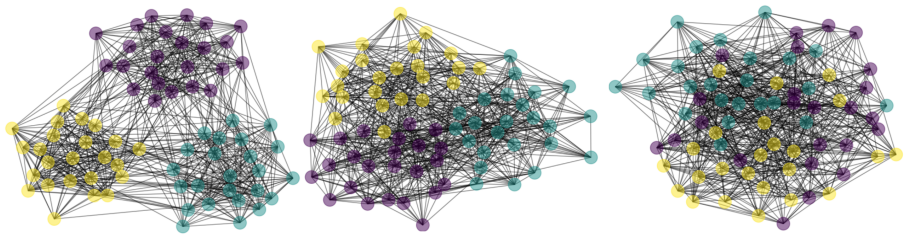

Figure 6: Visualization of GN benchamrk graphs with $\mu = [0.2, 0.4, 0.5]$

```
def girvan_newman(n=24, c=3, mu=0.2):
    G = nx.MultiGraph(name="Gir-New")
    for i in range(n*c):
        G.add_node(i, cluster=i//n+1) # add 3 groups/clusters of 24 nodes

    for i in range(len(G)):
        for j in range(i+1, len(G)):
            if G.nodes[i]['cluster'] == G.nodes[j]['cluster']:
                if random.random() < 20*(1-mu)/23: # add edge within the community
                    G.add_edge(i, j)
            else:
                if random.random() < 20*mu/48: # add edge outside the community
                    G.add_edge(i, j)
    #utils.info(G)
    return G
```

4.2 Lacichinetti benchmark graphs

Evaluating these results was the most tricky. Maybe I'm relying too much on visualizing clear community structures, but I feel that once the mixing parameter becomes too large (maybe bigger than 0.5), the graph looks completely random. The algorithms shouldn't detect communities in almost complete randomly mixed structures.

So in my opinion FLPA performed the best as it dropped the NMI the most as the mixing parameter μ increased. Second best Leiden and the worst Infomap. What possible community structure was detected in figure 8?

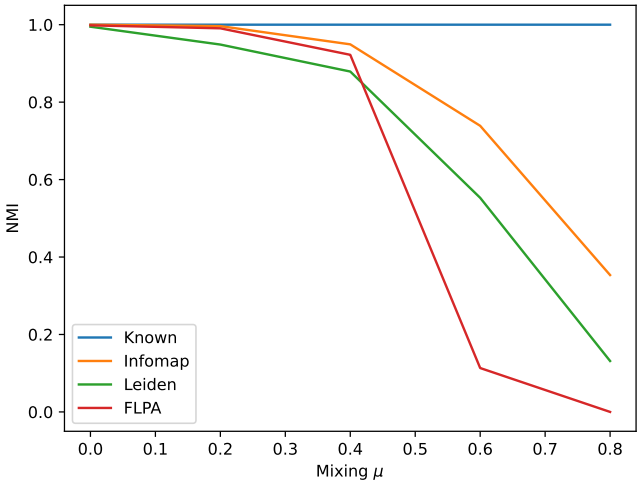

Figure 7: The accuracy of community detection algorithms (Infomap, Leiden, FLPA) on Lancichinetti benchmark graphs.

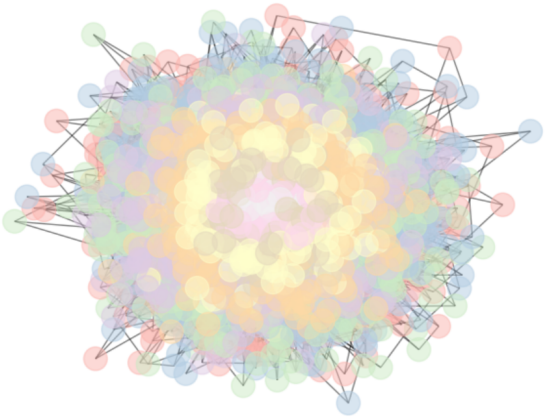

Figure 8: Example of Lacichinetti benchmark graph with mixing paramter $\mu = 0.8$

4.3 Erdos-Renyi random graphs

FLPA seems to be completely immune to random structure, it detected no communities or rather one single community through all variations of the ER graph (different average node degrees), so I'd say it performed the best. Infomap performed the worst at the start, but as we increased the average node degree it corrected itself to detecting no communities. Leiden performed the worst as it kept detecting communities (at least its consistent).

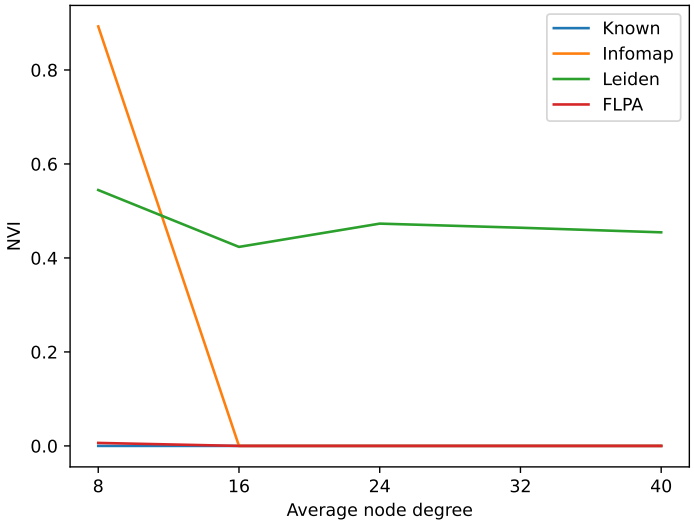

Figure 9: The accuracy of community detection algorithms (Infomap, Leiden, FLPA) on Erdos-Renyi random graphs.

5 Get at least 70% right!

Since achieving a classification accuracy of $\approx 67\%$ by simply looking at the majority class of the node's neighborhood suggests some degree of structure, we can likely enhance the results by considering the majority class of more coherent communities. The idea is to first use one of the community detection algorithms from the previous exercise, obtain the majority class of each cluster and then assign that class to all the nodes in the cluster.

I used fast label propagation (FLPA) as the community detection algorithm. Accuracy wise it was actually the second best (infomap being better by about 1%), but the compute time was a lot faster. The achieved average accuracy over the 10 runs was: ≈ 0.7363, which is $\approx 6\%$ better compared to simply using the nodes neighbourhood to predict its class (journal). The printout of the code is shown below.

```
G = u.read_pajek("aps_2008_2013.net")
u.info(G)

acc_sum = 0
for i in range(10):
    C = u.fast_label_propagation(nx.MultiGraph(G))
    #C = algorithms.leiden(G)
    #C = algorithms.infomap(G)

    data = pd.DataFrame()
    data['Nodes'] = list(G.nodes)
    data['GT'] = [int(G.nodes[node]['value']) for node in G.nodes]
    data['Predicted'] = 0

    for com in C.communities:
        count = {1: 0, 2: 0, 3: 0, 4: 0, 5: 0,
                 6: 0, 7: 0, 8: 0, 9: 0, 10: 0}
        for node in com:
            journal = G.nodes[node]['value']
            #print(int(journal))
            count[int(journal)] += 1
        max_key = max(count, key=count.get)
        data["Predicted"][com] = max_key

    acc = sum(data["GT"] == data["Predicted"])/len(data)
    acc_sum += acc
    print(f"Run {i} accuracy = {acc}")

print(f"Average classification accuracy over 10 runs: {acc_sum/10}")
```

6 Peers, ties and the Internet

6.1 No links predicted CA

Real networks are usually sparse networks, that means that the actual amount of links (links that form) is much less than the maximum possible number of links (links that could potentially form). If a method simply predicts that no links will occur, it would be correct for all the potential links that do not form, which is the majority. Therefore, this method would have a high classification accuracy.

However, this high accuracy is misleading. While the method would correctly predict most of the non-links, it would fail to predict any of the actual links that do form. It would have a high true negative rate but a zero true positive rate, which makes it a poor method for link prediction, despite its high overall accuracy. To properly evaluate a links prediction method, we need to consider other metrics that take into account false negatives (precision, recall, F1 score).

6.2 AUC results

Link Prediction Method / Network	Erdos-Renyi	Gnutella P2P	FB Circles	Nec Internet
Preferential Attachment Index	0.5	0.719	0.832	0.823
Adamic-Adar Index	0.5	0.514	0.994	0.696
Community Index	0.501	0.532	0.955	0.898

Table 2: Average AUC (over 10 runs) with different link prediction methods and networks.

The AUC for ER random graph seems as expected, around 0.5 across the board. As the links are attached randomly, predicting them can also be at best random. No method stands out, as it should be since Erdos-Renyi random graph has no structure that is biased towards any of the used methods.

Gnutella P2P network is a decentralized P2P network, which should include a bunch of random hubs, so the structure should gravitate towards a scale-free network. We can see that this seems to be true as preferential attachment index performed the best with an AUC of 0.72. Just generally all these networks other than ER random graph are real world networks, so they all have a certain degree of scale-free like structure. Preferential attachment index should be quite high for all of them (which it does).

Facebook circles network is a social circle network. It has high clustering (0.6) so the structure should gravitate towards small-world like. Adamic-Adar index (best for networks with small-world properties) as expected performed the best with an AUC of 0.994. Social networks should also have some reasonable community structure so Community index also performed really well (AUC = 0.955).

The Nec Internet map network is the one that gravitates towards community structures the most. We expect the internet communication to have a bunch of cluster/community like structures with a lot of internal connections and very few external connections. So as expected Community index performed the best with an AUC of 0.898.

6.3 Framework implementation

```
class GraphLinkPredictionMethods:
    def __init__(self, G, clusters=False) -> None:
        self.G = G
        if clusters:
            self.C = algorithms.leiden(G) # precompute clusters for the community index
            self.cluster_data, self.node_data = self.setup()

    def setup(self):
        cluster_data = {}
        node_data = {}

        for i, com in enumerate(self.C.communities):
            tmp_subgraph = self.G.subgraph(com)
            mc = tmp_subgraph.number_of_edges()
            nc = tmp_subgraph.number_of_nodes()
            s_ij = mc/math.comb(nc, 2) if nc >= 2 else 0

            cluster_data[i] = (mc, nc, s_ij)

            for node in com:
                node_data[node] = i

        return cluster_data, node_data

    def preferential_attachement_index(self, node1, node2): # preferential attachement index
        s_ij = self.G.degree(node1) * self.G.degree(node2)
        return s_ij

    def adamic_adar_index(self, node1, node2): # Adamic-Adar index
        neigh1 = set(self.G.neighbors(node1))
        neigh2 = set(self.G.neighbors(node2))
        common_neigh = neigh1.intersection(neigh2) # get common neighbors of node1 and node2
        s_ij = sum([1/np.log(self.G.degree(node)) for node in common_neigh])
        return s_ij

    def community_index(self, node1, node2):
        cluster_idx1 = self.node_data.get(node1)
        cluster_idx2 = self.node_data.get(node2)

        if cluster_idx1 is not None and cluster_idx1 == cluster_idx2:
            mc, nc, s_ij = self.cluster_data[cluster_idx1]
            return s_ij

        return 0

def get_negative_examples(G, examples):
    LN = set()
    nodes = list(G.nodes)
    while len(LN) < examples:
        # randomly sample the node pairs
        node1 = random.choice(nodes)
        node2 = random.choice(nodes)

        # check if they are neighbours
        if not G.has_edge(node1, node2):
            LN.add((node1, node2))

    return LN
```

```
def get_positive_examples(G, examples):
    G_changed = G.copy()
    LP = set()
    edges = list(G.edges)
    m = len(edges)

    while len(LP) < examples:
        idx = random.randint(0, m-1)
        edge = edges[idx]
        #edge = random.choice(edges)
        node1, node2 = edge
        LP.add(edge)
        G_changed.remove_edge(node1, node2)
        del edges[idx]
        m -= 1

    return LP, G_changed

def framework(G:nx.Graph, choosen_index:str):
    num_examples = G.number_of_edges() / 10

    #print("getting pos and neg node pairs...")
    LN = get_negative_examples(G, int(num_examples))
    LP, G_changed = get_positive_examples(G, int(num_examples))

    method_setup = GraphLinkPredictionMethods(G_changed, (choosen_index=="community_index"))
    index_method = getattr(method_setup, choosen_index)

    #print("calculating index for each pair...")
    # precompute the indexes for ALL the randomly selected pairs
    neg_indexes = [index_method(node1, node2) for node1, node2 in LN]
    pos_indexes = [index_method(node1, node2) for node1, node2 in LP]

    #print("sampling from calculated indexes...")
    # sample the indexes - m/10 times with repetitions
    neg_samples = [random.choice(neg_indexes) for _ in range(int(num_examples))]
    pos_samples = [random.choice(pos_indexes) for _ in range(int(num_examples))]

    m_prime = 0
    m_prime_prime = 0

    #print("calculating AUC...")
    for neg, pos in zip(neg_samples, pos_samples):
        if pos > neg:
            m_prime += 1
        elif pos == neg:
            m_prime_prime += 1

    auc = (m_prime + m_prime_prime/2) / num_examples
    return auc

def run_methods(G, n=10):
    print(f"Graph: {G.name}")
    for index_m in ["preferential_attachement_index", "adamic_adar_index", "community_index"]:
        aucs = []
        for _ in range(n):
            auc = framework(G, index_m)
            #print(auc)
            aucs.append(auc)
        print(f"Method {index_m} average (n=10) AUC: {round(sum(aucs)/n, 3)}")
```